A multi-method for load balancing with application to parallel dynamic programming algorithms for the 0-1 knapsack problem

M. Elkihel D. El Baz

LAAS du CNRS, 7, avenue du Colonel Roche, 31077 Toulouse CEDEX 4, France
E-mail: elkihel@laas.fr elbaz@laas.fr

Abstract

This paper deals with 0-1 knapsack problems. A load balancing multi-method is proposed and applied to a dynamic programming parallel algorithm using dominance technique and processor cooperation. Design of an efficient load balancing method is particularly important in this case since irregular data structure is generated by the parallel algorithm and the number of undominated states is unforeseeable. Computational experiments carried out on an Origin 3800 supercomputer are displayed and analyzed.

Key words: parallel computing, load balancing, combinatorial optimization, 0-1 knapsack problems, dynamic programming.

1 Introduction

The 0-1 knapsack problem which belongs to the class of NP-complete problems has been intensively studied in the literature (see for example [14], [3], [12], [19], [26], [24], [25], [21] and [23]). In this paper, we propose an efficient load balancing multi-method for a dynamic programming parallel algorithm using dominance technique and processor cooperation. Many study deal with dense dynamic programming, i.e. an approach for which one takes into account all possible states (see [1], [2], [4], [5], [7] and [15]). In this case, the number of states is equal to the capacity of the knapsack. In this paper, concentrate on a different approach proposed by Ahrens and Finke (see [3]) which permits one to limit the number of states by using dominance technique. In this case, the number of states or undominated pairs generated is unforeseeable. If we compare the two approaches, then we note that the former will generate a regular data stucture, from which one can easily deduce the total amount of work needed in order to treat the list. This approach leads generally to an easy parallelization; its main drawback is that it produces large lists in the

case of problems with large capacity. The later approach presents the advantage to generate lists which are smaller; its main drawback is that irregular data structure are created. As a consequence, the parallelization of the later approach is not easy and the design of an efficient load balancing strategy is very important.
In [11] we have presented an original parallelization of the one list dynamic programming method using dominance technique. Processors cooperation via data exchange is the main feature of the proposed parallel algorithm. A first load balancing strategy was also given in [11]. In this paper, we present an original load balancing multi-method.
We note that our contribution is different from the other works in the literature devoted to parallel dynamic programming for 0-1 knapsack problems. In particular, it is different from [6] and [18], where the authors have considered approaches based on massive parallelism. More precisely, in the above quoted papers, the authors have proposed solution for arrays with up to $O(2^{\frac{n}{8}})$ processors, where n is the number of variables in the knapsack problem. Our work is also different from [8], where the authors have studied the parallel implementation of a two lists algorithm on a MIMD architecture for the solution of a particular class of 0-1 knapsack problems: the exact subset sum problem where profits are equal to weights. In this later approach, total work is decomposed initially and processors do not cooperate. Note that our parallel algorithm is designed for a broad class of 0-1 knapsack problems including subset sum problems. Moreover, our parallel algorithm presents better time complexity than the parallel algorithm studied in [8], as we shall see in the sequel. Reference is also made to [7] and [13] for different approaches concerning the parallelisation of the dynamic programming method.
Section 2 deals with the 0-1 knapsack problem and its solution via dynamic programming. The parallel algorithm is presented in Section 3. An original load balancing multi-method is proposed in Section 4. Finally, computational results carried out on an Origin 3800 supercomputer are displayed and analyzed in Section 5.

2 The 0-1 Knapsack Problem

The 0-1 unidimensional knapsack problem is defined as follows:

$$\max\left\{\sum_{j=1}^{n} p_j x_j | \sum_{j=1}^{n} w_j x_j \leq C \ ; x_j \in \{0,1\}, j = 1, 2, \ldots, n\right\}, \tag{1}$$

where C denotes the capacity of the knapsack, n the number of items considered, p_j and w_j, respectively, the profit and weight, respectively, associated with the j-th item. Without loss of generality, we assume that all the data are positive integers. In order to avoid trivial solutions, we assume that we have: $\sum_{j=1}^{n} w_j > C$ and $w_j < C$ for all $j \in \{1, ..., n\}$. Several methods have been proposed in order to solve problem (1). We can quote for example: branch and bound methods proposed by Fayard and Plateau (see [12]), Lauriere (see [17]) and Martello and Toth (see [19]), methods based on dynamic programming studied by Horowitz and Sahni (see [14]), Ahrens

and Finke (see [3]) and Toth (see [27]) and finally mixed algorithms combining dynamic programming and branch and bound methods presented by Martello and Toth (see [20]) and Plateau and Elkihel (see [26]).
In this paper, we concentrate on a dynamic programming method proposed by Ahrens and Finke whose time and space complexity are $O(\min\{2^n, nC\})$ (see [3]) and which is based on the concepts of list and dominance. We shall generate recursively as follows lists L_k of pairs (w, p), $k = 1, 2, \ldots, n$; where w is a weight and p a profit. Initially, we have $L_0 = \{(0, 0)\}$.
Let us define the set N_k of new pairs generated at stage k; where new pairs results from the fact that a new item, i.e. the k-th item, is taken into account.

$$N_k = \{(w + w_k, p + p_k) \mid (w, p) \in L_{k-1},\ w + w_k \leq C\}. \tag{2}$$

According to the dominance principle, which is a consequence of Bellman's optimality principle, all pairs $(w, p) \in L_{k-1} \cup N_k$ obtained by construction such that there exists a pair $(w', p') \in L_{k-1} \cup N_k$, $(w', p') \neq (w, p)$, which satisfies: $w' \leq w$ and $p \leq p'$, must not belong to a list L_k. In this case, we usually say that the pair (w', p') dominates the pair (w, p). As a consequence, any two pairs $(w', p'), (w'', p'')$ of the list L_k must satisfy: $p' < p''$ if $w' < w''$. Thus, we can define the set D_k of dominated pairs at stage k as follows.

$$\begin{aligned} D_k = \{(w, p) \mid (w, p) \in L_{k-1} \cup N_k, \exists (w', p') \in L_{k-1} \cup N_k \text{ with} \\ w' \leq w, p \leq p', (w', p') \neq (w, p)\}. \end{aligned} \tag{3}$$

As a consequence, for all positive integers k, the dynamic programming recursive list L_k is defined as follows

$$L_k = L_{k-1} \cup N_k - D_k. \tag{4}$$

Note that the lists L_k are organized as sets of monotonically increasing ordered pairs in both weight and profit. As a consequence, the largest pair of the list L_n, corresponds to the optimal value of the knapsack problem.

3 Parallel Algorithm

In this Section, we detail the parallelization of the one list dynamic programming method using dominance technique. The parallel algorithm which was brievly presented in [11] is designed according to the SPMD model for a parallel architecture that can be viewed as a shared memory machine on a logical point of view. As we shall see in detail in Section 5 experiments have been carried out on a NUMA (non uniform memory access) supercomputer Origin 3800 by using the Open MP environment.

The main feature of this parallel algorithm is that all processors cooperate via data exchange to the construction of the global list. The global list is partitioned into sublists. Sublists are organized as sets of monotonically increasing ordered pairs in both weight and profit. Each sublist is generated by one processor of the parallel architecture. In particular, all dominated pairs are removed at each stage of the parallel dynamic programming method. More precisely, at stage k, each processor E^i generates a sublist of the global list L_k, which is denoted by L_k^i. The total work is shared by the different processors and data exchange permits each processor to remove all dominated pairs from its sublist based on global information. It is important to keep in mind that parallel algorithms whereby dominated pairs are not removed at each stage do not correspond to a parallel implementation of the dynamic programming using dominance technique algorithm, but in the special case of the subset sum problem where profits are equal to weight. Moreover, in the general case, the approach whereby all dominated pairs are not removed from the sublists may not be efficient, since the number of dominated pairs may be huge. The benefit of parallelism can then be lost since parallel algorithms will have to deal with a greater number of pairs than the sequential dynamic programming algorithm using dominance technique.
Several issues must be addressed when considering the design of a parallel algorithm: initialization of the parallel algorithm, work decomposition, tasks assignation, data exchange and load balancing strategy.

3.1 Initialization, Work Decomposition and Task Assignment

The initialization of the parallel algorithm is performed by a sequential dynamic programming algorithm using dominance technique. First of all, a sequential process performs $k(0)$ stages of the dynamic programming algorithm. This process generates a list which contains at least lq pairs, where q denotes the total number of processors and l the minimal number of pairs per processor. Let $L_{k(0)}$ be the ordered list which results from the sequential initialization. The list $L_{k(0)}$ is partitionned as follows: $L_{k(0)} = \cup_{i=0}^{q-1} L_{k(0)}^i$, with $\mid L_{k(0)}^i \mid = l$, for $i = 1, ..., q-1$ and $\mid L_{k(0)}^0 \mid \geq l$, where $\mid L_{k(0)}^i \mid$ denotes the number of pairs of the sublist $L_{k(0)}^i$.
If all processors E^i generate independently their sublist L_k^i without sharing data with any other processor, then, on a global point of view, some dominated pairs may belong to the sublists L_k^i, which induces finally an overhead. Thus, in the beginning of each stage, it is necessary to synchronize all the processors which must then share part of the data produced at the previous stage in order to discard all dominated pairs in a global way. The use of global dominance technique can reduce the cardinality of the sublists L_k^i. At each stage, the contents of the union of the generated sublists is the same as the contents of the list generated by a sequential dynamic programming algorithm using dominance technique as we shal see in detail in the next subsection.

3.2 Parallel processes

We detail now the parallel algorithm. We present first the process of construction of the sublists L_k^i generated at each stage. We introduce the various sublists that are created by the different processors E^i, $i = 0, ..., q-1$, at each stage k in order to generate the sublists L_k^i. For all $i \in \{0, ..., q-1\}$, the smallest pair of L_k^i in both weight and profit will be denoted by $(w_k^{i,0}, p_k^{i,0})$. The various sublists are defined as follows. For all $i = 0, ..., q-2$,

$$N_k^i = \{(w_{k-1}^{i,\cdot} + w_k, p_{k-1}^{i,\cdot} + p_k) \mid (w_{k-1}^{i,\cdot}, p_{k-1}^{i,\cdot}) \in L_{k-1}^i, w_{k-1}^{i,\cdot} + w_k < w_{k-1}^{i+1,0}\}, \quad (5)$$

and

$$N_k^{q-1} = \{(w_{k-1}^{q-1,\cdot} + w_k, p_{k-1}^{q-1,\cdot} + p_k) \mid (w_{k-1}^{q-1,\cdot}, p_{k-1}^{q-1,\cdot}) \in L_{k-1}^{q-1}, w_{k-1}^{q-1,\cdot} + w_k \leq C\}. (6)$$

The sublist N_k^i corresponds to the new list of pairs created at stage k by processor E^i from its sublist L_k^{i-1} and the k-th item and which are assigned to processor E^i. Some pairs clearly do not belong to the sublist N_k^i, i.e. the pairs for which the weight $w_{k-1}^{i,\cdot} + w_k$ is greater than or equal to the weight $w_{k-1}^{i+1,0}$ of the smallest pair of the list L_{k-1}^{i+1} generated by processor E^{i+1}. Those discarded pairs which are stored as shared variables are used by processors E^j with $i < j$, in order to generate their sublists L_k^j as we shall see in the sequel. It is important to note at this point that for all $i \in \{0, ..., q-1\}$, data exchange can then occur only between processor E^i and processors E^j with $i < j$. For this purpose, consider now the series of sets C_k^i defined as follows. For all $i \in \{1, ..., q-2\}$,

$$\begin{gathered} C_k^i = \{(w_{k-1}^{j,\cdot} + w_k, p_{k-1}^{j,\cdot} + p_k) \mid (w_{k-1}^{j,\cdot}, p_{k-1}^{j,\cdot}) \in L_{k-1}^j, j < i, \\ w_{k-1}^{i,0} \leq w_{k-1}^{j,\cdot} + w_k < w_{k-1}^{i+1,0} \text{ or } w_{k-1}^{j,\cdot} + w_k < w_{k-1}^{i,0}, p_{k-1}^{j,\cdot} + p_k \geq p_{k-1}^{i,0}\}, \end{gathered} \quad (7)$$

$$C_k^0 = \emptyset, \quad (8)$$

and

$$\begin{gathered} C_k^{q-1} = \{(w_{k-1}^{j,\cdot} + w_k, p_{k-1}^{j,\cdot} + p_k) \mid (w_{k-1}^{j,\cdot}, p_{k-1}^{j,\cdot}) \in L_{k-1}^j, j < q-1, \\ w_{k-1}^{q-1,0} \leq w_{k-1}^{j,\cdot} + w_k < C \text{ or } w_{k-1}^{j,\cdot} + w_k < w_{k-1}^{q-1,0}, p_{k-1}^{j,\cdot} + p_k \geq p_{k-1}^{q-1,0}\}. \end{gathered} \quad (9)$$

The sublist C_k^i is in fact the set of pairs that are exchanged between all processors E^m, with $m < i$ and processor E^i, either in order to complete the sublist L_k^i that will be produced by processor E^i at stage k or to permit processor E^i to discard from its sublist some dominated pairs, at stage k. This last decision being made on a global point of view. In particular, it is important to note that all processors E^j, with $j > i$, must share with processor E^i all the pairs created by E^i that will permit E^j to eliminate dominated pairs. In order to discard all the pairs which must not belong to the sublist L_k^i and particularly dominated pairs, we introduce the series of sets D_k^i. For all $i = 0, ..., q-2$,

$$D_k^i = \hat{D}_k^i \cup \{(w,p) \mid w < w_{k-1}^{i+1,0} \text{ and } p \geq p_{k-1}^{i+1,0}\}, \tag{10}$$

with

$$\begin{aligned}\hat{D}_k^i = \{(w,p) \mid (w,p) \in L_{k-1}^i \cup N_k^i \cup C_k^i \text{ and } \exists (w',p') \in L_{k-1}^i \cup N_k^i \cup C_k^i, \\ (w',p') \neq (w,p) \text{ and } w' \leq w, p \leq p'\},\end{aligned} \tag{11}$$

and

$$\begin{aligned}D_k^{q-1} = \{(w,p) \mid (w,p) \in L_{k-1}^{q-1} \cup N_k^{q-1} \cup C_k^{q-1}, \exists (w',p') \in L_{k-1}^{q-1} \cup N_k^{q-1} \cup C_k^{q-1}, \\ (w',p') \neq (w,p), w' \leq w, p \leq p'\},\end{aligned} \tag{12}$$

We note that $\hat{D}_k^i$ is the subset of D_k^i which contains all dominated pairs in processor E^i at stage k. A similar remark can be made for the set D_k^{q-1}. We note also that copies of the same pair may be present in in different processors. These copies permit processors to check dominance relation. All copies must be removed at each stage but one, i.e. the last one. As a consequence, the dynamic programming recursive sublists L_k^i are defined as the following sets of monotonically increasing ordered pairs in both weight and profit. For all positive integer k and all $i = 0, ..., q-1$,

$$L_k^i = L_{k-1}^i \cup N_k^i \cup C_k^i - D_k^i. \tag{13}$$

Initially, note that we have $L_{k(0)} = \cup_{i=0}^{q-1} L_{k(0)}^i$, as it was stated in the beginning of subsection 3.1. Assume now that for a given k, we have $L_{k-1} = \cup_{i=0}^{q-1} L_{k-1}^i$, then it follows clearly from the definition of N_k, N_k^i and C_k^i that $\cup_{i=0}^{q-1}(N_k^i \cup C_k^i) = N_k$, i.e. no pair is lost in the construction and data exchange processes. Moreover it follows from the definitions of $\hat{D}_k^i$ and D_k^{q-1} that $\cup_{i=0}^{q-2} \hat{D}_k^i \cup D_k^{q-1} = D_k$, i.e. no dominated pair remains in the sublists L_k^i. We note that the elimination of all dominated pairs in each processor is not local. The elimination is rather based on global information. Thus, it follows from (4) and (13) that $\cup_{i=0}^{q-1} L_k^i = L_k$.

We present now the parallel dynamic programming algorithm designed according to the Single Program Multiple Data (SPMD) model. The parallel algorithm was carried out on a nonuniform memory access (NUMA) shared memory Origin 3800 supercomputer by using the Open MP environment. More precision on the machine can be found in Section 5. All the variables are local, otherwise it is said in the algorithm.

Parallel Algorithm

FOR $k = k(0) + 1$ TO n
DO
 FOR $i = 0$ TO $q - 1$
 DO IN PARALLEL
  BEGIN
   IF $i \neq q - 1$
   THEN
    IF the pair $(w_{k-1}^{i+1,0}, p_{k-1}^{i+1,0})$, is available
    THEN
     BEGIN
      E^i generates N_k^i;
      E^i stores as shared variables all pairs $(w_{k-1}^{i,\cdot} + w_k, p_{k-1}^{i,\cdot} + p_k) \mid$
      $(w_{k-1}^{i,\cdot}, p_{k-1}^{i,\cdot}) \in L_{k-1}^i$, $w_{k-1}^{i+1,0} \leq w_{k-1}^{i,\cdot} + w_k \leq C$;
      E^i stores as shared variables all pairs $(w_{k-1}^{i,\cdot} + w_k, p_{k-1}^{i,\cdot} + p_k) \mid$
      $(w_{k-1}^{i,\cdot}, p_{k-1}^{i,\cdot}) \in L_{k-1}^i$, $w_{k-1}^{i,\cdot} + w_k < w_{k-1}^{i+1,0}$, $p_{k-1}^{i,\cdot} + p_k \geq p_{k-1}^{i+1,0}$;
     END
   ELSE
    E^{q-1} generates N_k^{q-1};
   BARRIER OF SYNCHRONIZATION;
   E^i generates C_k^i;
   E^i generates D_k^i;
   E^i generates L_k^i;
   IF $i \neq 0$
   THEN
    E^i stores $(w_k^{i,0}, p_k^{i,0})$ as a shared variable;
  END

We note that all processors E^j, $j = 1, ..., q-1$, store the smallest pair $(w_{k-1}^{j,0}, p_{k-1}^{j,0})$ of their sublist L_{k-1}^j as a shared variable since this value is used by processors E^i with $i < j$ to determine what pairs must be stored as a shared variable in order to be exchanged with E^j. We note also that pair exchange between processors occurs always between a processor E^i and a processor E^j, with $j > i$. Thus, processor E^{q-1} plays a particular part with this type of data exchange since, generally, it tends to accumulate more pairs than any other processor.

The particular data structure chosen in order to store exchanged pairs as shared variables is a table with three entries. Typically, processor E^i will read data exchanged in i lists which are generated, respectively by processors E^0 to E^{i-1}.

4 Load Balancing

In order to obtain good performance, it is necessary to design an efficient load balancing strategy. As a matter of fact, if no load balancing technique is implemented, then it results in particular from the data exchange process described in the previous Section that processor E_q can become overloaded.

In this Section, we propose a multi-method which combines two load balancing strategies which are both designed in order to obtain a good efficiency. When using load balancing the time complexity of the parallel algorithm presented in Section 3 is $O(\min\{\frac{2^n}{q}, \frac{nC}{q}\})$, since the number of pairs will be fairly distributed on the different processors. We note that both strategies used are different from the one considered in [11], which is an adaptive strategy whereby a load balancing is made if any processor which is overloaded can take benefit of it and decisions are taken every two stages according to measures and estimations of the load. For more details concerning the latter strategy, the reader is referred to [11]. We present now the two strategies used in the load balancing multi-method.

4.1 Dynamic load balancing

The first strategy considered in this paper does not necessarily balance loads at each stage; this strategy is based upon a test which is made at each stage. The load balancing test is based upon a comparison of the work needed for performing the load balancing on the one hand and the work resulting from the load unbalancing on the other hand. The later work is related to the difference of number of pairs between the largest sublist and the other lists. If the load balancing work is more expensive than the work needed for processing pairs, then the loads are not balanced, otherwise they are balanced. The load balancing process will assign fairly loads to processors, i.e. it will give approximatively an equal number of pairs to all processors as we shall see in what follows.

In the sequel, T_p, T_w and T_r, respectively, denote the processing time, the writing time and the reading time relative to one pair, respectively. At any given stage k, the number of pairs of the largest sublist is denoted by N_l and the total number of pairs assigned to all processors is denoted by N_t. The load unbalancing cost which is denoted by c_u is given by

$$c_u = T_p.(N_l - \frac{N_t}{q}). \tag{14}$$

The load balancing cost which is denoted by c_b is given by

$$c_b = N_l.(T_r + T_w), \tag{15}$$

since read and write are made in parallel in each processor. Thus, the test will be basically as follows. If $c_u > c_b$, then we balance loads, else the loads are not balanced. The test can be rewritten as follows

$$N_l - \frac{N_t}{q} > N_l . \frac{(T_r + T_w)}{T_p}, \tag{16}$$

which can also be rewritten

$$1 - \frac{N_t}{q.N_l} > \frac{(T_r + T_w)}{T_p}, \tag{17}$$

or

$$1 - \frac{(T_r + T_w)}{T_p} > \frac{N_t}{q.N_l}. \tag{18}$$

In the next Section, we will present computational experiments carried out on the Origin 3800 parallel supercomputer. We have obtained the following measurements on the Origin 3800 for T_p, T_r and T_w.

$$T_p = 2.69.10^{-7} \text{ s}, \; T_r = 4.2.10^{-8} \text{ s and } T_w = 3.6.10^{-8} \text{ s}. \tag{19}$$

Thus, for this machine we have

$$\frac{(T_r + T_w)}{T_p} = 0.29, \tag{20}$$

and the practical test is given as follows.

$$0.71 > \frac{N_t}{N_l . q}. \tag{21}$$

The reader is referred to [22] for dynamic load balancing approaches which present some similarities with our dynamic strategy and which are applied to adaptive grid-calculations for unsteady three-dimensional problems. However, we note that our test (16) is different from the test used in [22] (reference is also made to [9] and [16]).

In the next subsection, we present a different load balancing strategy which is also performant and simple.

4.2 Implicit load balancing

Implicit load balancing has been designed in order to decrease overhead while performing a fair load balancing. The principle of implicit load balancing is very simple. Since the capacity C of the knapsack is given and the size of the lists L_k is at most equal to C, the idea is to assign to processor E^0 the pairs with weight between 0 and $\lfloor \frac{C}{q} \rfloor$ where q denotes the number of processors, similarly, processor E^1 will be assigned the pairs with weight between $\lfloor \frac{C}{q} \rfloor + 1$ and $2.\lfloor \frac{C}{q} \rfloor$ and so on.

The main advantage of this strategy is that pairs are directly assigned to a given processor according to their weight. As a consequence, there is no overhead like in the case of the previous strategy. On the other hand, the main drawback of this strategy is its inefficiency at initialization, since idle times of some processors such as for example E^{q-1} or E^{q-2} may be nonnegligeable. However, we will see in the next Section that this load balancing technique is performant even when the number of processors is large. As a matter of fact, with these technique, the work of the different processors tends to be well distributed, since according to the strategy, the number of pairs assigned to all processors tends to be constant in working regime.

We present now the multi-method.

4.3 Load balancing multi-method

In this subsection, we propose a multimethod which combines the advantages of dynamic and implicit load balancing. More precisely, we will use dynamic load balancing in the beginning of the treatment since it must perform better at that time than implicit load balancing. We recall that some processors are inactive in the beginning with implicit load balancing while they are all active with dynamic load balancing. Then, we shall end treatment with implicit load balancing, since it presents no overhead, contrarily to dynamic load balancing.

The major questions at this point are: at what time must we swap methods and what is the expected gain in combining dynamic and implicit load balancing as compared with the performence of both methods taken solely?

We propose an automatic procedure which swaps load balancing methods. This procedure is based on the fact that implicit load balancing will be very efficient when total work will be shared by the different processors; i.e. when processor E^{q-1} will start to be assigned pairs with weight between $(q-1)\lfloor \frac{C}{q} \rfloor + 1$ and C. In the sequel $(w_k^{q-1,\omega}, p_k^{q-1,\omega})$. will denote the largest pair of L_k^{q-1} in both weight and profit.

The following algorithm shows how the load balancing multi-method works.

Load balancing multi method

$k = k(0) + 1$
WHILE $w_k^{q-1,\omega} \leq (q-1)\lfloor \frac{C}{q} \rfloor$ and $k \leq n$
DO
 BEGIN
 Perform dynamic load balancing
 $k = k + 1$
 END
IF $k \leq n$
THEN
 DO
  FOR $j = k$ TO n
   BEGIN
   Perform implicit load balancing
   END

We note that in some cases the multi-method may only implement dynamic load balancing. This is particularly true if the number of processors is large; in that case $(q-1)\lfloor \frac{C}{q} \rfloor$ may be very close to C and there is no guarantee that even for a large k there exists a pair with weight $w_k^{q-1,\omega}$ greater than or equal to $(q-1)\lfloor \frac{C}{q} \rfloor$.

5 Computational Experiments

The numerical experiments presented in this Section correspond to weakly correlated problems. These problems can be considered as difficult (see [10]). We have avoided treating noncorrelated problems which induce a large number of dominated pairs and which are thus easier to solve.

The various instances considered are relative to problem sizes which are equal to 200, 400, 1000 and 5000, respectively, with data range defined as follows. The weights w_j are randomly distributed in the segment $[1, 1000]$ and the nonnegative profits p_j in $[w_j - g, w_j + g]$, where the gap, denoted by g, is equal to 100. We have considered problems such that $C = 0.5 \sum_{j=1}^{n} w_j$; this permits one to generate difficult instances.

The parallel algorithm has been implemented in C on a NUMA (non uniform memory access) shared memory supercomputer Origin 3800, using the Open MP environment. The architecture of the machine is hybrid, i.e. there is some shared and distributed memory. However, total memory can be viewed as shared on a logical point of view. More precisely, the parallel architecture is an hypercube constituted by 512 processors MIPS R14000 with 500 MHz clock frequency and 500 Mo of memory per processor. The operating system is IRIX 6.5 IP35. The total bandwith of the communication network is 716 Go/s. We note that generally, all processors do not have the same read or write time for every value. With this architecture, the read or write time in a remote part of the memory may be 2 or 3 times greater than the read or write time in the local memory of a processor. However, we have made computational tests with 2.10^6 up to 8.10^7 pairs; in this range we have noted a constant duration of 4.10^{-8} seconds for the read time as well as the write time.

Parallel numerical experiments have been carried out with up to 32 processors. Numerical results are displayed on Tables 1 to 3. In Tables 1 to 3, we give the running time in seconds for sequential algorithms, which is denoted by t_s, and parallel algorithms, denoted by t_p. Times correspond to an average taken over 20 instances. Experiments are reported for 4, 8, 16 and 32 processors We also give the average efficiency of parallel algorithms which is equal to $\frac{t_s}{q.t_p}$. We note from Tables 1 to 3 that the efficiency of the parallel algorithm is function of several parameters such as the size of the problem or the number of processors. We note that the efficiency of the parallel dynamic programming algorithm using the multi-method, dynamic or implicit load balancing tends generally to decrease when the number of processors increases. Basically, for these problems the granularity, i.e. the ratio computation time over communication time, tends to decrease when q increases, since in this case communications play a prominent part.

Parallel algorithms with dynamic or implicit load balancing strategy present in general a good efficiency for a coarse granularity. The performance of these two load balancing strategies is also better than the one of the strategy presented in one of our previous papers (see [11]). The performance of implicit load balancing is quite similar to the performance of dynamic load balancing; although, the more sophisticated dynamic load balancing strategy seems a little bit more performant than implicit load balancing for large size problems and large number of processors. We see also that the performance of the multi method is in general better than the one of dynamic or implicit load balancing. This last remark is particularly true for large size problems. This shows the interest of our approach. We note also that in some cases, efficiency may be greater than one. Experiments were carried out on a non uniform memory architecture. Thus, several processors can use more efficiently their fast local memory, where local data are stored, then a single processor which needs to access sometimes remote part of the memory in order to use all the data of the problem. We recall that the access to remote part of the memory costs more than the access to local memory on a NUMA architecture.

q	1	4		8		16		32	
size	t_s	t_p	e	t_p	e	t_p	e	t_p	e
200	0.116	0.056	0.52	0.047	0.31	0.059	0.12	0.102	0.04
400	0.775	0.28	0.69	0.18	0.54	0.15	0.32	0.19	0.13
1000	7.935	2.56	0.77	1.51	0.66	0.92	0.54	0.69	0.36
5000	463.48	103.78	1.12	63.86	0.91	33.48	0.86	17.59	0.82

Table 1. Computing time in seconds and efficiency
for a gap 100 and range 1000 with load balancing multi method

q / size	1	4		8		16		32	
size	t_s	t_p	e	t_p	e	t_p	e	t_p	e
200	0.116	0.064	0.45	0.049	0.3	0.06	0.12	0.109	0.03
400	0.775	0.32	0.60	0.2	0.48	0.15	0.32	0.21	0.11
1000	7.935	3.15	0.63	1.68	0.59	0.97	0.51	0.75	0.33
5000	463.48	140.57	0.82	68.02	0.85	33.76	0.86	18.3	0.79

Table 2. Computing time in seconds and efficiency
for a gap 100 and range 1000 with dynamic load balancing

q / size	1	4		8		16		32	
size	t_s	t_p	e	t_p	e	t_p	e	t_p	e
200	0.116	0.06	0.49	0.04	0.35	0.04	0.21	0.04	0.09
400	0.775	0.32	0.60	0.19	0.52	0.12	0.4	0.11	0.22
1000	7.935	3.09	0.64	1.68	0.59	0.95	0.52	0.64	0.39
5000	463.48	130.27	0.89	67.18	0.86	34.80	0.83	18.78	0.77

Table 3. Computing time in seconds and efficiency
for a gap 100 and range 1000 with implicit load balancing

Acknowledgments

Part of this study has been made possible by a grant of CINES, Montpellier, France.

References

[1] R. Andonov and F. Gruau, A linear systolic array for the knapsack problem, *Proceedings of the International Conference on Application Specific Array Processors* **23**, 458-472 (1991).

[2] R. Andonov and P. Quinton, Efficient linear systolic array for the knapsack problem, *in L. Bougé, et al. editors Parallel Processing: CONPAR 92 - VAPP V, Lecture Notes in Computer Science, Springer Verlag* **634**, 247-258 (1992).

[3] J. H. Ahrens and G. Finke, Merging and Sorting Applied to the Zero-One Knapsack Problem, *Operations Research* **23**, 1099-1109 (1975).

[4] H. Bouzoufi and S. Boulenouar and R. Andonov, Pavage optimal pour des dépendances de type sac à dos, *Actes du Colloque RenPar'10* (1998).

[5] J. Casti and M. Richardson and R. Larson, Dynamic programming and Parallel Computers, *Journal of Optimization Theory and Applications* **12**, 423-438 (1973).

[6] H. K. C. Chang and J. J. R. Chen and S. J. Shyu, A parallel algorithm for the knapsack problem using a generation and searching technique, *Parallel Computing*, **20** 233-243 (1994).

[7] G. H. Chen and M. S. Chern and J. H. Jang, Pipeline Architectures for Dynamic Programming Algorithms, *Parallel Computing* **13**, 111-117 (1990).

[8] M. Cosnard and A. G. Ferreira and H. Herbelin, The two lists algorithm for the knapsack problem, *Parallel Computing* **9**, 385-388 (1989).

[9] G. Cybenko, Dynamic load balancing for distributed-memory multiprocessors *Journal of Parallel an Distributed Computing* **7**, 279-301 (1989).

[10] M. Elkihel, Programmation dynamique et rotation de contraintes pour les problèmes d'optimisation entière, *Thèse de Doctorat, Université des Sciences et Techniques de Lille*, (1984).

[11] M. Elkihel and D. El Baz, An efficient dynamic programming parallel algorithm for the 0-1 knapsack problem, in Joubert et al. (eds) *Parallel Computing Advances and Current Issues* Imperial College Press, London, 298-305 (2002).

[12] D. Fayard and G. Plateau, Resolution of the 0-1 Knapsack Problem: Comparison of methods, *Mathematical Programming* **8**, 272-307 (1975).

[13] T. E. Gerash and P. Y. Wang, A survey of parallel algorithms for one-dimensional integer knapsack problems, *INFOR* **32**, 163-186 (1994).

[14] E. Horowitz and S. Sahni, Computing partitions with application to the knapsack problems, *Journal of the Association for Computing Machinery* **21**, 275-292 (1974).

[15] G. Kindervater and J.K. Lenstra, An introduction to parallelism in combinatorial optimization, *Discrete and Applied Mathematics*, **14** 135-156 (1986).

[16] G. Kohring, Dynamic load balancing for parallelized particle simulations on MIMD computers *Parallel Computing*, **21** 638-693 (1995).

[17] M. Lauriere, An Algorithm for the 0-1 knapsack problem, *Mathematical Programming* **14**, 1-10 (1978).

[18] D. C. Lou and C. C. Chang, A parallel two list algorithm for the knapsack problem, *Parallel Computing*, **22**, 1985-1996 (1997).

[19] S. Martello and P. Toth, An Upper bound for the Zero One Knapsack Problem and a Branch and Bound Algorithm, *European Journal of Operational Research* **1**, 169-175 (1977).

[20] S. Martello and P. Toth, Algorithm for the Solution of the 0-1 single Knapsack Problem, *Computing* **21**, 81-86 (1978).

[21] S. Martello and D. Pisinger and P. Toth, Dynamic programming and strong bounds for the 0-1 knapsack problem, *Management Science* 45, 414-424 (1999).

[22] L. Oliker and R. Biswas, PLUM: Parallel load balancing for adaptive unstructured meshes, *Journal of Parallel and Distributed Computing* 52, 150-157 (1998).

[23] U. Pferschy, Dynamic programming revisited: Improving knapsack algorithms, *Computing* 63, 419-430 (1999).

[24] D. Pisinger, A minimal algorithm for the 0-1 knapsack problem, *Operations Research* **45**, 758-767 (1997).

[25] D. Pisinger and P. Toth, Knapsack problems, in D. Z. Du and P. Pardalos (eds) *Handbook of Combinatorial Optimization*, Kluwer, 1-89 (1998).

[26] G. Plateau and M. Elkihel, A Hybrid Method for the 0-1 Knapsack Problem, *Methods of Operations Research* **49**, 277-293 (1985).

[27] P. Toth, Dynamic Programming Algorithms for the Zero-One Knapsack Problem, *Computing* **25**, 29-45 (1980).